Neural Networks and Function Approximation

Krishna Kumar

University of Texas at Austin

krishnak@utexas.edu

Overview

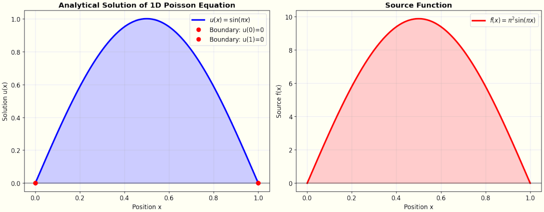
Outline

The 1D Poisson Equation: Our Benchmark Problem

Consider the one-dimensional Poisson equation on $[0, 1]$:

$$-\frac{d^2u}{dx^2} = f(x), \quad x \in [0, 1]$$

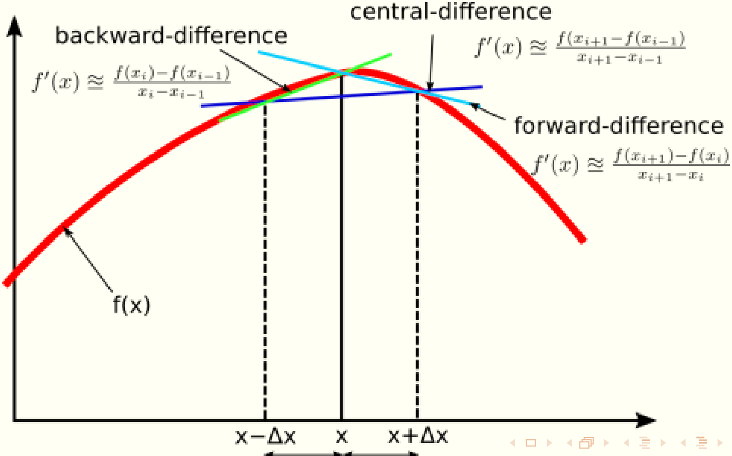
with boundary conditions: $u(0) = 0, \quad u(1) = 0$
Chosen source term: $f(x) = \pi^2 \sin(\pi x)$
Analytical solution: $u(x) = \sin(\pi x)$



Analytical solution to 1D Poisson equation

Finite difference approach:

- Discretize domain: $x_i = i\Delta x$, $i = 0, 1, ..., N$
- Approximate derivatives: $u''(x_i) \approx \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$
- Solve linear system: $Au = f$



Neural Network Approach

Different paradigm: Learn a continuous function $u_{NN}(x; \theta)$ that approximates $u(x)$.

- Network is a parameterized function approximator
- Train on sample points from the domain
- Evaluate anywhere (not just at grid points)
- Leverage automatic differentiation for derivatives

Key question: Can neural networks approximate arbitrary continuous functions?

Outline

The Perceptron: From Linear to Nonlinear

Step 1: Linear Perceptron

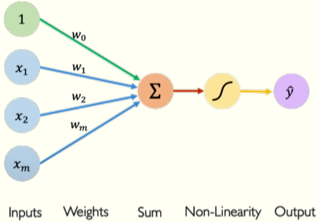
$$\hat{y} = \mathbf{w}^T\mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

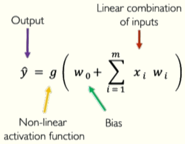
This is just a linear transformation - can only model linear relationships.

Step 2: Add Activation Function

$$z = \mathbf{w}^T\mathbf{x} + b \quad \text{(pre-activation)}$$
$$\hat{y} = g(z) \quad \text{(output)}$$

The activation function g introduces nonlinearity.





Why We Need Nonlinearity

Linear Limitation: A linear perceptron creates a hyperplane decision boundary.

Linearly Separable

- Single line can separate classes
- Linear perceptron works

Not Linearly Separable

- No single line works
- Need nonlinear boundaries

Example: Circular Pattern

Points inside circle (class 0) vs outside (class 1) - impossible for linear boundary.

Solution: Activation functions enable learning curved decision boundaries.

Common Activation Functions

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

- Output in $(0, 1)$
- Smooth gradient
- Can saturate

ReLU: $\text{ReLU}(z) = \max(0, z)$

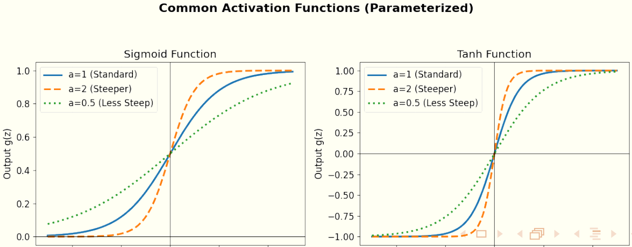
- Simple and efficient
- No saturation for $z > 0$
- Dead neurons for $z < 0$

Tanh: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

- Output in $(-1, 1)$
- Zero-centered
- Can saturate

Leaky ReLU: $\max(\alpha z, z)$

- Prevents dead neurons
- Small slope for $z < 0$



Training: Gradient Descent

Objective: Minimize loss function $L = \frac{1}{2N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$
Gradient Computation (Single Sample):
For prediction $\hat{y} = \mathbf{w}^T \mathbf{x} + b$ and loss $L = \frac{1}{2}(y - \hat{y})^2$:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \mathbf{w}} = -(y - \hat{y}) \cdot \mathbf{x}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = -(y - \hat{y})$$

Update Rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} + \eta (y - \hat{y}) \mathbf{x}$$
$$b \leftarrow b - \eta \frac{\partial L}{\partial b} = b + \eta (y - \hat{y})$$

where η is the learning rate.

Backpropagation with Activation Functions

With activation g, the forward pass: $z = \mathbf{w}^T\mathbf{x} + b$, $\hat{y} = g(z)$

Chain Rule for Gradients:

1. **Output error:** $\delta = \frac{\partial L}{\partial \hat{y}} = -(y - \hat{y})$
2. **Pre-activation gradient:** $\frac{\partial L}{\partial z} = \delta \cdot g'(z)$
3. **Weight gradient:** $\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial z} \cdot \mathbf{x} = \delta \cdot g'(z) \cdot \mathbf{x}$
4. **Bias gradient:** $\frac{\partial L}{\partial b} = \delta \cdot g'(z)$

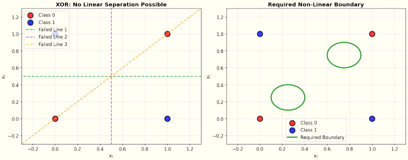
Implementation:

The XOR Problem: Motivation for Deep Networks

XOR Truth Table:

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

The Problem: No single line can separate the classes.



XOR data points

Key Insight:

- Single perceptron fails (even with nonlinearity)
- Need multiple layers
- Hidden layer learns intermediate features
- Output layer combines features

Historical Impact: Led to first AI winter (Minsky & Papert, 1969)

Multi-Layer Networks: Solution to XOR

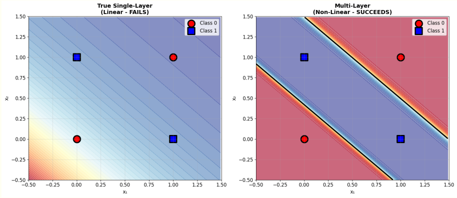
Two-layer network can solve XOR:

Hidden Layer:

- Neuron 1: Detects $(x_1 = 1, x_2 = 0)$
- Neuron 2: Detects $(x_1 = 0, x_2 = 1)$

Output Layer:

- Combines hidden features
- OR operation on detectors



How hidden layer transforms XOR

Key Concept: Each layer transforms data into new representation where it becomes more linearly separable.

Outline

Universal Approximation Theorem

Theorem (Cybenko, 1989): A single hidden layer network with sufficient neurons can approximate any continuous function to arbitrary accuracy.

$$F(x) = \sum_{i=1}^{N} w_i \sigma(v_i x + b_i) + w_0$$

Formal Statement: For any continuous $f : [0,1] \to \mathbb{R}$ and $\epsilon > 0$, there exists N and parameters such that:

$$|F(x) - f(x)| < \epsilon \quad \forall x \in [0,1]$$

Important Caveats:

- Theorem guarantees existence, not how to find parameters
- May need exponentially many neurons
- Deeper networks often more efficient in practice

Single Hidden Layer Architecture

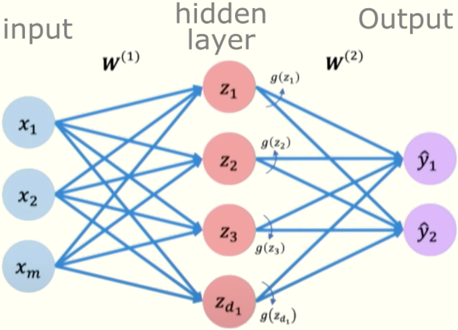
For 1D input x, network with N_h hidden neurons:

Forward Pass:

$$\mathbf{z}^{(1)} = W^{(1)}x + \mathbf{b}^{(1)}$$
$$\mathbf{h} = g(\mathbf{z}^{(1)})$$
$$z^{(2)} = W^{(2)}\mathbf{h} + b^{(2)}$$
$$\hat{y} = z^{(2)}$$

where:

- $W^{(1)} \in \mathbb{R}^{N_h \times 1}$
- $W^{(2)} \in \mathbb{R}^{1 \times N_h}$



Single hidden layer network

Outline

The Training Process

Objective: Find parameters $\theta^* = \arg\min_\theta \mathcal{L}(\theta)$
Loss Function (MSE):

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(u_{NN}(x_i;\theta) - u_i)^2$$

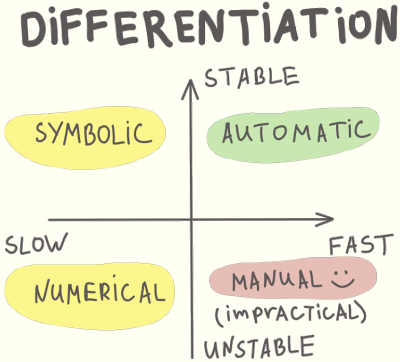
Training Algorithm:

1. **Initialize:** Random weights (small values)
2. **Forward Pass:** Compute predictions $\hat{y}_i$
3. **Compute Loss:** Evaluate $\mathcal{L}(\theta)$
4. **Backward Pass:** Compute gradients via backpropagation
5. **Update:** $\theta \leftarrow \theta - \eta\nabla_\theta\mathcal{L}$
6. **Repeat:** Until convergence

Training neural networks requires gradients: $\frac{\partial \mathcal{L}}{\partial \theta}$ for all parameters θ.

Traditional approaches have fundamental flaws:

- **Manual:** Exact, but error-prone and doesn't scale
- **Symbolic:** Exact, but "expression swell"
- **Numerical:** $f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$
  - Inaccurate (rounding errors)
  - Expensive (multiple evaluations)



For a neural network with thousands of parameters:

$$\theta = \{W^{(1)}, \mathbf{b}^{(1)}, W^{(2)}, \mathbf{b}^{(2)}, \ldots\}$$

We need a fourth approach: Automatic Differentiation!

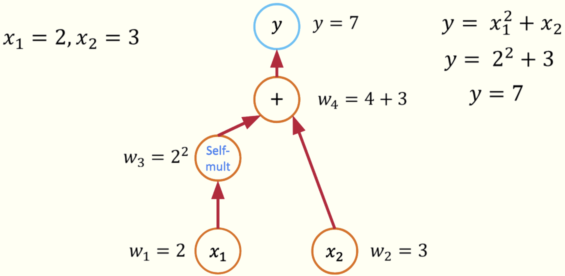
Automatic Differentiation: The Solution

Every function is a computational graph of elementary operations.

Consider: $y = x_1^2 + x_2$

Evaluation Trace:

1. $v_1 = x_1^2$
2. $y = v_1 + x_2$

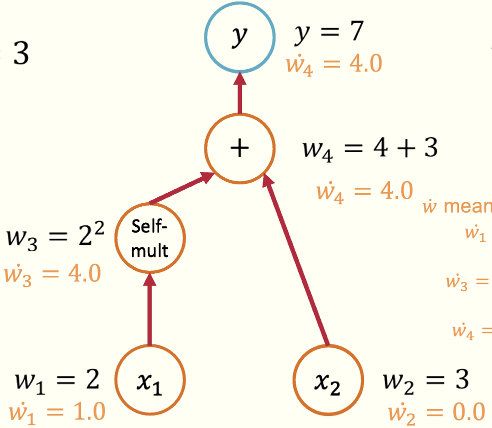


This decomposition is the key that makes AD possible. By applying the chain rule to each elementary step, we can compute exact derivatives.

Forward Mode Automatic Differentiation

Forward mode AD propagates derivative information **forward** through the graph, alongside the function evaluation.

$$x_1 = 2, x_2 = 3$$



$$y = x_1^2 + x_2$$

$$y = 2^2 + 3$$

$$y = 7$$

$\dot{w}$ means differentiating w.r.t x_1

$\dot{w}_1 = 1.0$ and $\dot{w}_2 = 0.0$

$$\dot{w}_3 = 2w_1 \cdot \dot{w}_1 = 2(2).1 = 4$$

$$\dot{w}_4 = \dot{w}_3 + \dot{w}_2 = 4 + 0 = 4$$

$$\dot{y} = \dot{w}_4 = 4.0$$

AD Demo

Reverse Mode Automatic Differentiation (Backpropagation)

Key Insight: For scalar loss functions, reverse mode computes all gradients in one pass.

Algorithm:

1. **Forward Pass:** Evaluate function, store intermediate values
2. **Backward Pass:** Propagate gradients backward using chain rule

Example for Perceptron:

$$\text{Forward:} \quad z = \mathbf{w}^T\mathbf{x} + b, \quad a = g(z), \quad L = \frac{1}{2}(y - a)^2$$

$$\begin{aligned}
\text{Backward:} \quad & \frac{\partial L}{\partial a} = -(y - a) \\
& \frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot g'(z) \\
& \frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial z} \cdot \mathbf{x} \\
& \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z}
\end{aligned}$$

When to Use Forward vs. Reverse Mode

The choice depends on the dimensions of your function $f : \mathbb{R}^n \to \mathbb{R}^m$.

Forward Mode

- Cost: One pass per input
- Efficient when: $n \ll m$
- Few inputs, many outputs

Reverse Mode

- Cost: One pass per output
- Efficient when: $n \gg m$
- Many inputs, few outputs

Neural Networks: Millions of parameters (inputs), single scalar loss (output)

Reverse mode (backpropagation) wins!

AD Notebook

Basic SGD:

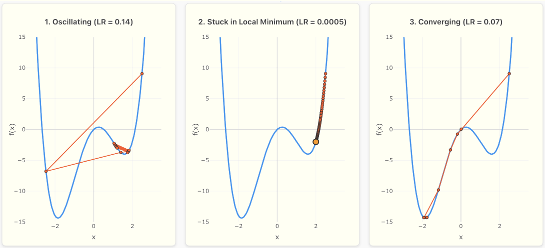
$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}$$

SGD with Momentum:

$$v_{t+1} = \beta v_t + \nabla_\theta \mathcal{L}$$
$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$

Adam (Adaptive Moments):

- Adapts learning rate per parameter
- Combines momentum + RMSprop
- De facto standard in deep learning

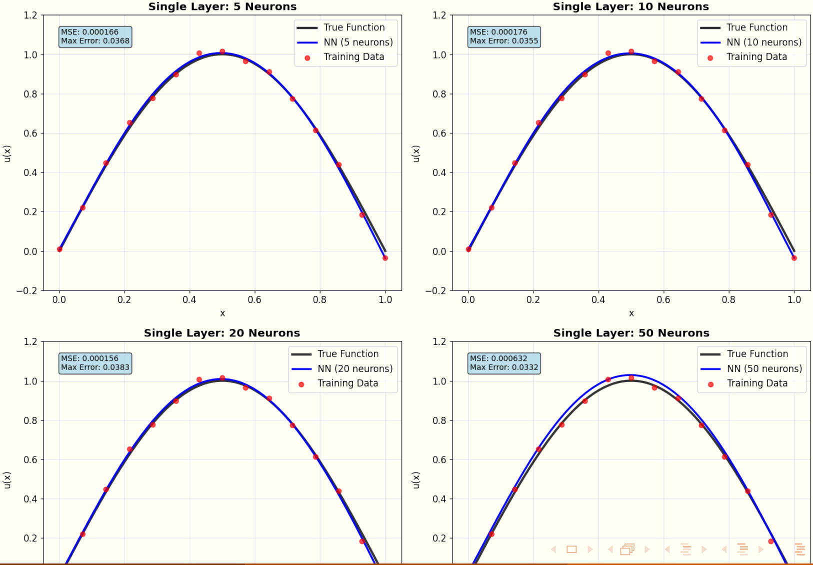


Gradient descent trajectory

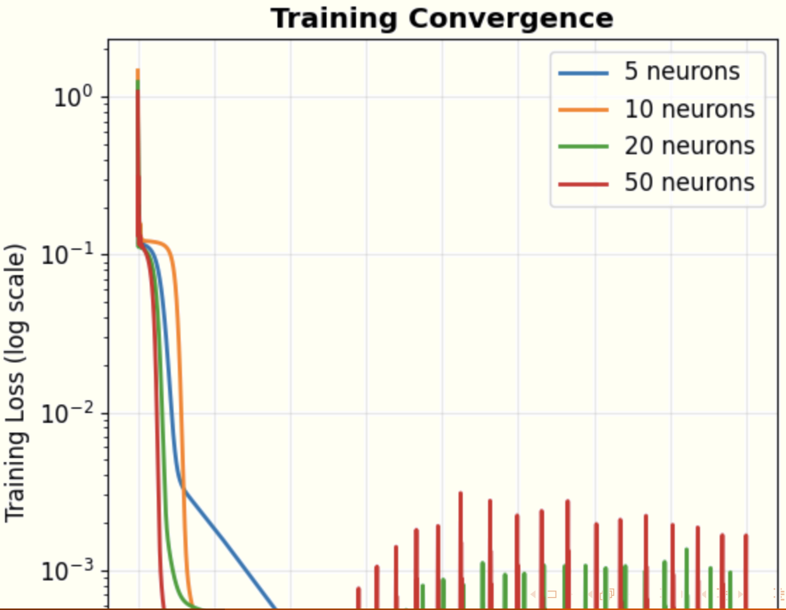
Outline

Width vs. Approximation Quality

Experiment: How does network width affect approximation quality?



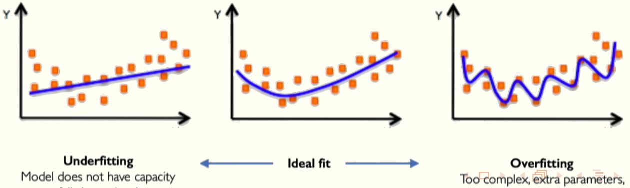
Training Dynamics



Outline

Common Training Challenges

1. **Vanishing/Exploding Gradients**
   - Deep networks compound gradients
   - Solution: Careful initialization, normalization
2. **Local Minima**
   - Non-convex optimization
   - Solution: Multiple runs, momentum
3. **Overfitting**
   - Network memorizes training data
   - Solution: Regularization, dropout
4. **Computational Cost**
   - Many parameters to optimize
   - Solution: GPUs, efficient algorithms



Outline

Analogy with Numerical Methods:

Neural Networks	**Numerical PDEs**
Weights $\mathbf{w}$	Solution coefficients
Loss function L	Residual $\|\mathcal{L}[u] - f\|$
Gradient descent	Iterative solver
Learning rate η	Time step/relaxation parameter
Activation functions	Basis functions

Physics-Informed Neural Networks (PINNs):
Minimize combined loss:

$$L(\theta) = \underbrace{\|\mathcal{L}[u_{NN}] - f\|_{\Omega}^2}_{\text{PDE loss}} + \lambda \underbrace{\|u_{NN} - g\|_{\partial\Omega}^2}_{\text{Boundary loss}}$$

Network learns to satisfy both PDE and boundary conditions.

Summary

Key Concepts:

- **Perceptron:** Linear transformation + activation
- **Nonlinearity:** Essential for complex functions
- **Training:** Gradient descent with backpropagation
- **Universal Approximation:** Single layer can approximate any continuous function
- **Automatic Differentiation:** Enables efficient gradient computation

Practical Insights:

- Start with linear model, add complexity as needed
- XOR problem motivates deep networks
- More neurons improve approximation (with diminishing returns)
- Connection to traditional numerical methods

Next Steps: Deep networks, advanced architectures, PINNs

Resources

Interactive Demos:

- ReLU Activation: Demo
- SGD Visualization: Demo
- AD Graph: Demo

Jupyter Notebooks:

- Complete MLP implementation: GitHub
- Automatic Differentiation: Notebook

References:

- Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function"
- Hornik, K. (1991). "Approximation capabilities of multilayer feedforward networks"
- Goodfellow et al. (2016). "Deep Learning" (MIT Press)